Lecture 16: Linear Algebra III

Instructor: Professor Amos Ron Scribes: Mark Cowlishaw, Nathanael Fillmore

For the last few lectures we have discussed numerical solutions to linear systems of equations

$$A\vec{x} = \vec{b}$$

We have particularly focused on four fundamental topics:

Algorithms

We introduced two direct algorithms for solving linear systems, both using the technique of *factorization*: LU Factorization, and QR-factorization. Recall that LU-factorization factors a matrix into the product of an upper-triangular and lower-triangular matrix. QR-factorization factors a matrix into the product of an orthogonal and upper triangular matrix.

Complexity/Cost

We stated last time that for a system of n equations in n unknowns (represented by an $N \times N$ matrix A), both LU and QR-factorization take time proportional to the cube of n ($O(n^3)$). LU-factorization takes less time by a constant factor of around 2 - as n gets large, this constant factor is dwarfed by the third power of the matrix dimension, however.

Note that the theoretical "best-case" for the complexity of solving a linear system is $O(n^2)$, since there are n^2 entries in the matrix A, and any correct algorithm would have to examine each entry.

Conditioning

As we discussed last time, *conditioning* is a property of the matrix A that determines whether we can expect *any* numeric algorithm to provide a numerical solution to a linear system involving A. The idea is that any linear system we represent on a computer has some error, since we always use a fixed precision, that is the linear equation

$$A\vec{x} = \vec{b}$$

really becomes a slightly perturbed system when represented on a computer:

$$\hat{A}\hat{\vec{x}} = \hat{\vec{b}}$$

So that, if A has the property that small perturbations in $\vec{b}$ can produce large changes in $\vec{x}$, we have no hope of solving any linear system involving A, no matter which algorithm we use. Recall that we measure this property of A using the condition number:

$$cond(A) = \frac{\max\{\|A\vec{v}\| : \vec{v} \in \mathbb{C}^n \wedge \|\vec{v}\| = 1\}}{\min\{\|A\vec{v}\| : \vec{v} \in \mathbb{C}^n \wedge \|\vec{v}\| = 1\}}$$

If the condition number is large for a matrix A, then no system of linear equations

$$A\vec{x} = \vec{b}$$

can be reliably solved using *any* numerical algorithm. For example, recall the ill-conditioned matrix we discussed in the last lecture, and the difference in the norms of vectors $(A\vec{x})$ as measured using the infinity norm:

$$\begin{aligned}
& \left\| \begin{bmatrix} 1 & 1 \\ 1 & 1+\epsilon \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\|_\infty \\
= & \left\| \begin{bmatrix} 2 \\ 2+\epsilon \end{bmatrix} \right\|_\infty \\
= & \; 2+\epsilon \\
& \left\| \begin{bmatrix} 1 & 1 \\ 1 & 1+\epsilon \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\|_\infty \\
= & \left\| \begin{bmatrix} 0 \\ -\epsilon \end{bmatrix} \right\|_\infty \\
= & \; \epsilon
\end{aligned}$$

Note that, since the norm of the second vector is very, very small, the condition number of the matrix will be quite large, so that we cannot reliably solve any equation involving this matrix using any numerical algorithm.

Stability

Unlike conditioning, stability is a property of the numerical algorithm and not of the system of equations. Assuming that we are given a *good* linear system, stability measures how reliably a particular algorithm will solve it.

For example, consider solving the following linear system of equations using brute-force elimination (as we will see, this is essentially LU-factorization):

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(1) \\ x(2) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

As shown in Figure 1, the angle between the two equations:

$$\begin{aligned}
\epsilon \cdot x(1) + x(2) &= 1 \quad \text{(and)} \\
x(1) + x(2) &= 2
\end{aligned}$$

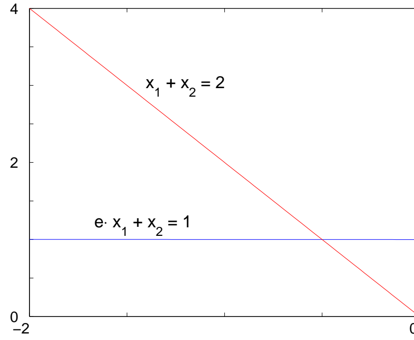

Figure 1: A Well-Conditioned Linear System

is quite large, so that we can see that the system is well-conditioned. Furthermore, since $\epsilon \ll 1$ the solution $x(1) = x(2) = 1$ is nearly correct. If we solve this system using brute-force elimination, we would multiply the top row by $1/\epsilon$ and subtract it from the bottom row:

$$\begin{bmatrix} \epsilon & 1 \\ 0 & 1 - 1/\epsilon \end{bmatrix} \cdot \begin{bmatrix} x(1) \\ x(2) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 - 1/\epsilon \end{bmatrix}$$

Using back-substitution, we see that:

$$\begin{aligned} x(2) &= \frac{2 - 1/\epsilon}{1 - 1/\epsilon} \\ &\sim \frac{-1/\epsilon}{-1/\epsilon} \quad (2)\ (\text{since } 1/\epsilon >> 1, 2) \\ &\sim 1 \end{aligned}$$

Which leads to the solution $x(1) = 0, x(2) = 1$, which is clearly not the solution to the original equation. Note that the round-off that happens at step (2) depends only on ϵ. If ϵ is small enough, it will cause this rounding in any fixed precision representation of numbers.

There is also another more theoretical way to see how our algorithm is at fault in the example above. As was already hinted, what we did above was actually to solve the system via an LU-factorization $A = LU$: We first solved the L-system

$$Ly = b$$

where

$$L = \begin{bmatrix} 1 & 0 \\ \frac{1}{\epsilon} & 1 \end{bmatrix}$$

and we then solved the U-system

$$Ux = y$$

where

$$U = \begin{bmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{bmatrix}.$$

In order for this approach to avoid instability, the condition numbers of both L and U need to be small: if L is ill-conditioned, then the method may fail while solving the L-system, while if U is ill-conditioned, then the method may fail while solving the U-system. In our example above, $\text{cond}(L) \approx 1$, but $\text{cond}(U) \gg 1$, and indeed the algorithm introduces error precisely when solving the U-system.

So if LU-factorization is unstable, why does Matlab use it? The answer is that the instability of LU-factorization can be avoided by permuting the rows of the matrix A. For example, we could rewrite the previous system with the first and second rows exchanged:

$$\begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix} \cdot \begin{bmatrix} x(1) \\ x(2) \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Elimination produces the following upper triangular system:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 - \epsilon \end{bmatrix} \cdot \begin{bmatrix} x(1) \\ x(2) \end{bmatrix} = \begin{bmatrix} 2 \\ 1 - 2/\epsilon \end{bmatrix}$$

This system avoids the huge factor $1 - 1/\epsilon$, and hence does not lead to the round-off error we saw in the previous system. We call this reordering of rows a *pivot*. Pivoting can always be used to make LU-factorization a stable algorithm.

Matlab uses pivoting to stabilize LU-factorization when you call `lu`:

```
[L,U,P] = lu(A)
```

Here `P` is a permutation matrix which permutes the rows of `A` in order to avoid an unstable LU-factorization.

Note that QR-factorization is stable without pivoting, and takes roughly twice as long as LU-factorization. (LU without pivoting costs $\sim n^3/6$ multiplications, while QR costs $\sim n^3/3$ multiplications.) Thus, any calculations to determine pivots in LU-factorization must take less than a constant factor of 2 in the overall running time to make LU-factorization cost-effective. We know that such efficient pivoting algorithms must exist, since Matlab uses QR-factorization as the default direct method for solving linear systems of equations.

1 Indirect Methods

So far, we have discussed only direct algorithms for solving linear systems of equations. As we have seen, the cost of these algorithms is $O(n^3)$ for an $n \times n$ matrix A. As the dimension n of the matrix A gets very large, the problem of solving the system may become computationally

infeasible, due to this cubic term. For example, a matrix with dimension $n = 10^4$ would take roughly $O((10^4)^3) = O(10^{12})$ computations to solve.

However, if we look at the theoretical minimum of $O(n^2)$, a system of size $n = 10^4$ might be solvable in a reasonable period of time $O((10^4)^2) = O(10^8)$. This gap between systems that can be reasonably solved using an algorithm with cost $O(n^2)$ but not an $O(n^3)$ algorithm is filled by so-called indirect methods.

Indirect methods use an iterative process, similar to the iterative methods we used for solving equations earlier this semester. The idea is to produce a series of solutions:

$$\vec{x_0}, \vec{x_1}, \ldots, \vec{x_k}$$

with each transition $\vec{x_i} \mapsto \vec{x_{i+1}}$ taking less than $O(n^2)$ operations. If $k \ll n$, that is, if we can reach an acceptable solution in a relatively small number of iterations, then this method will be more efficient than a direct method.

Indirect methods have an additional advantage - it turns out that as matrices get very large, small calculation errors are magnified during factoring, so that there is a practical limit to how large a linear system can be solved by direct methods.

2 Rectangular Systems

So far we have discussed the solution of square linear systems - systems in which the number of equations match the number of unknowns. We will now look at solving rectangular linear systems, systems in which the number of equations is larger than the number of unknowns (overdetermined systems), or is smaller than the number of unknowns (underdetermined systems).

2.1 Overdetermined Linear Systems

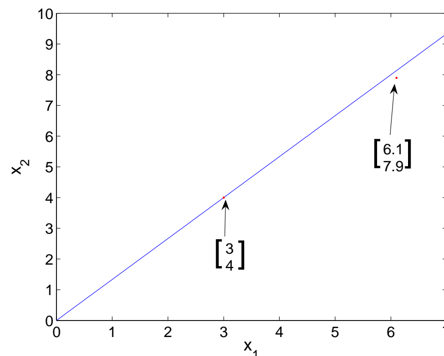

Figure 2: An Overdetermined Linear System

An overdetermined system is a system of linear equations in which the number of equations is larger than the number of unknowns. For example, consider the following linear system:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} x(1) = \begin{bmatrix} 6.1 \\ 7.9 \end{bmatrix}$$

Clearly, this system has no exact solution, as depicted in Figure 2, so what do we mean when we ask for a solution to the system? Intuitively, what we would like is the solution with the smallest amount of error. If we define the error for an overdetermined linear system

$$A\vec{x} = \vec{b}$$

using the difference between the vector $(A\vec{x})$ and the vector $\vec{b}$, i.e.,

$$e = A\vec{x} - \vec{b},$$

we notice that the error e is a vector, and we can measure its size using a norm. So, we solve an overdetermined system $A\vec{x} = \vec{b}$ by minimizing $\left\|A\vec{x} - \vec{b}\right\|$ for some norm. The choice of norm is important, as it may have a profound effect on the solution.

2.2 Least Squares

If we measure success using the 2-norm, then we are said to be using a *least squares* method.

For example, if we solve an overdetermined system as in the previous section using the 2-norm to measure our error, then we are using a least squares method to solve our overdetermined system. More formally, given an overdetermined linear system

$$A\vec{x} = \vec{b},$$

solving this system using least squares means finding a vector x^* with the following property:

$$\left\|A\vec{x^*} - \vec{b}\right\|_2 \leq \left\|A\vec{x} - \vec{b}\right\|_2 \text{ for all vectors } \vec{x}$$

Geometrically, we can think of the least squares method as finding the point on the vector $(A\vec{x})$ at the minimum Euclidean distance from $\vec{b}$, as shown in Figure 3.

It turns out that there is a simple formula for solving overdetermined systems using least squares, we simply multiply by the transpose of the matrix A on the left:

$$\begin{aligned} A_{m\times n}\vec{x}_{n\times 1} &= \vec{b}_{m\times 1} \quad (m > n) \\ A'_{n\times m}\left(A_{m\times n}\vec{x}_{n\times 1}\right) &= A'_{n\times m}\vec{b}_{m\times 1} \\ (A'A)_{n\times n}\vec{x}_{n\times 1} &= (A'\vec{b})_{n\times 1} \end{aligned}$$

Note that the last equation is a square linear system in which $(A'A)$ has dimension n. To illustrate, consider our example from the last section:

$$\begin{aligned} \begin{bmatrix} 3 \\ 4 \end{bmatrix} x(1) &= \begin{bmatrix} 6.1 \\ 7.9 \end{bmatrix} \\ \left([3 \quad 4] \begin{bmatrix} 3 \\ 4 \end{bmatrix} \right) x(1) &= [3 \quad 4] \begin{bmatrix} 6.1 \\ 7.9 \end{bmatrix} \\ [25]\, x(1) &= [49.9] \\ x(1) &\approxeq 1.996 \end{aligned}$$

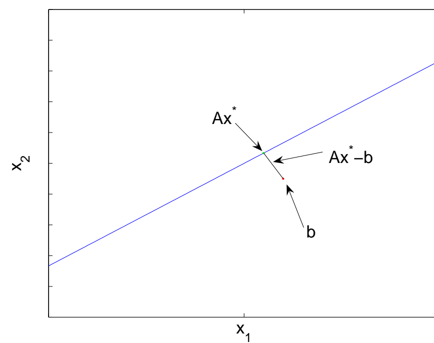

Figure 3: Solution to an Overdetermined Linear System using the 2-Norm